Desenvolvimento de uma Linguagem PicoC

Ricardo Lucena
Departamento Informática
Universidade do Minho
Braga, Portugal
PG54187@alunos.uminho.pt

José Fonte
Departamento Informática
Universidade do Minho
Braga, Portugal
A91775@alunos.uminho.pt

Daniel Du
Departamento Informática
Universidade do Minho
Braga, Portugal
PG53751@alunos.uminho.pt

***Abstract*— O seguinte relatório irá analisar o processo de desenvolvimento sobre uma linguagem *PicoC*, desde a criação da sua gramática, até ao desenvolvimento de programas que devolvam resultados específicos após serem interpretados.**

***Index terms*—Haskell, QuickCheck, Mutações, Processamento de Linguagens, Interpretador**

I. Introduction

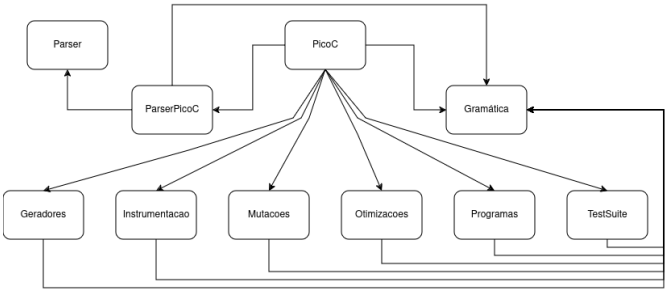
No contexto do desenvolvimento de linguagens de programação, a criação de uma nova linguagem representa um desafio considerável. Este trabalho apresenta um estudo detalhado sobre a concepção e implementação da linguagem *PicoC*, uma linguagem projetada com objetivos específicos de simplicidade e eficiência.

O desenvolvimento deste projeto passou por etapas como a criação da gramática inicial, e do *parser* e *unparser*, até ao testar da robustez da linguagem e do interpretador, que são avaliados através de mutações e técnicas de *fault localization*.

Este trabalho, desenvolvido no âmbito da disciplina de Manutenção e Evolução de Software, não só detalha cada uma das etapas por que passou, mas também reflete sobre os desafios enfrentados e as soluções implementadas. A criação do *PicoC* ilustra um processo completo de desenvolvimento de uma linguagem, desde a teoria até a prática.

II. Estrutura

Na figura abaixo está representada a estrutura do nosso programa. Decidimos dividir por módulos de forma a mantermos o código organizado e de que seja mais fácil de editar em caso de necessidade.



III. Gramática

A baixo apresentamos a gramatica para a nossa linguagem PicoC. Para além da gramática base facultada pela unidade docente, acrescentamos alguns parametros nos datatypes de forma a possuirmos uma linguagem mais completa e rica. Por exemplo, no datatype *Inst* acrescentamos o *COMS, PRINT e Eliminar* e no *Exp* acrescentamos *Great, GreatEqual, Minor*, entre outros.

```
data PicoC = PicoC [Inst]
  deriving (Data, Eq, Typeable)

data Inst = Atrib String Exp
          | While Exp BlocoC
          | ITE   Exp BlocoC BlocoC
          | COMS String
          | PRINT Exp
          | Eliminar
    deriving (Eq, Data, Typeable)

type BlocoC = [Inst]
type Inputs = [(String, Int)]

data Exp = Add Exp Exp
         | Mul Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
         | Const Int
         | Var   String
         | Great Exp Exp
         | GreatEqual Exp Exp
         | Minor Exp Exp
         | MinorEqual Exp Exp
         | Equals Exp Exp
         | Dif Exp Exp
```

```
        | Not Exp
        | And Exp Exp
        | Or Exp Exp
        | Verdadeiro
        | Falso
        | Str String
    deriving (Eq, Data, Typeable)
```

IV. Otimizações

Um dos componentes críticos deste trabalho é o módulo de otimizações, que visa melhorar a eficiência do código gerado sem alterar seu comportamento.

Por exemplo, porque é que nos iriamos contentar por ter a seguinte linha de código:

```
x = 2 + 0;
```

Quando é mais legível e eficiente ter só:

```
x = 2;
```

Este é apenas uma das muitas otimizações efetuadas pelo nosso programa, com ajuda da biblioteca *StrategicData*.

As otimizações básicas implementadas no módulo incluem simplificações de expressões aritméticas, como a eliminação de adições, como mostramos anteriormente, e a simplificação de subtrações e divisões triviais. Tudo isso é feito nas seguintes funções:

```
opt :: Exp -> Maybe Exp
optAritmeticas :: Exp -> Maybe Exp
optBooleanas :: Exp -> Maybe Exp
```

Mais à frente no trabalho prático fomos criando outras otimizações, por exemplo, o eliminar de comentários.

Após as otimizações estarem escritos, era necessário aplicá-las. Para esse fim, utilizamos a biblioteca *Zipper*, como podemos ver no exemplo abaixo de uma das funções responsável por aplicar um tipo de otimizações:

```
estBooleanas :: PicoC -> PicoC
estBooleanas p =
    let pZipper = toZipper p
        Just newP = applyTP (innermost  step) pZipper
        step = failTP `adhocTP` optBooleanas
    in fromZipper newP
```

Esta função aplica as otimizações definidas em *optBooleanas* ao código, utilizando a função *applyTP* para aplicar transformações de forma recursiva e abrangente.

Finalmente, a função *megaEstrat* combina várias estratégias de otimização para aplicar uma série de transformações:

```
megaEstrat :: PicoC -> PicoC
megaEstrat p = estTrue $ estBooleanas $ estAritmeticas
$ estAritmeticasfraca p
```

V. Geradores

O módulo Geradores foi criado para facilitar a geração automática de programas na linguagem PicoC, utilizando a biblioteca *QuickCheck*.

No caso da nossa linguagem, o nosso objetivo final era ter uma função que fosse capaz de gerar um programa completo *PicoC*. No entanto, para isso acontecer teríamos de avançar por partes, criando primeiro geradores simples de variáveis ou de inteiros e avançar posteriormente para partes mais complicadas.

Assim, foram criadas as seguintes funções:

```
genConst :: Gen Exp
genBoolean :: Gen Exp
genVar :: Gen Exp
genNot :: Int -> Gen Exp
genConds :: Int -> Gen Exp
genOrdGrandeza :: Int -> Gen Exp
genContas :: Int -> Gen Exp
genContas2 :: Int -> Gen Exp
genIgualdade :: Gen Exp
```

Todas estas funções foram posteriormente agrupadas na *genExp*, uma função responsável por gerar uma expressão:

```
genExp :: Int -> Gen Exp
genExp 0 = frequency [  (30, genConst),
                        (30, genVar),
                        (10, genBoolean)
                     ]
genExp n = frequency [  (30, genConst),
                        (30, genVar),
                        (10, genNot n),
                        (10, genBoolean),
                        (10, genConds n),
                        (10, genOrdGrandeza n),
                        (30, genContas n),
                        (10, genIgualdade)
                     ]
```

Nesta função fornecemos um inteiro *n* para limitar o número de parcelas criadas pelo gerador. Quando o *n* chega a 0 simplesmente dizemos para ele criar algo simples como um número, uma variável, ou um booleano, em vez de criar algo que possa dar continuidade a uma expressão maior, como uma soma.

Com o gerador de expressões finalizado, o próximo passo seria criar um gerador de instruções. Para isso foram criadas as seguintes funções:

```
genAtrib :: Int -> Gen Inst
genWhile :: Int -> Gen Inst
genITE :: Int -> Gen Inst
genComs :: Gen Inst
genInst :: Int -> Gen Inst
genBlocoC :: Int -> Gen BlocoC
genPicoC :: Int -> Gen PicoC
```

As últimas duas funções já tomam proveito do nosso gerador de instruções, de forma a alcançar o nosso objetivo final de gerar um programa *PicoC*, enquanto que todas as outras são

responsáveis por criar os diferentes tipos de instruções que temos no nosso programa.

Também utilizamos o *shrinking* que é uma característica importante do *QuickCheck*. Esta tenta minimizar um caso de teste que falha, tornando mais fácil identificar o problema. Implementamos instâncias de *Arbitrary* para expressões, instruções e programas, incluindo funções de shrinking que geram versões menores e mais simples das estruturas.

VI. Interpretador

Nesta secção do relatório iremos abordar a forma como resolvemos a primeira alínea do enunciado do trabalho prático, que pedia para criarmos uma função *evaluate*, que recebe um programa *PicoC* e uma lista de *Inputs*, e devolve o resultado final.

A forma como nós fazemos para obter o resultado final é através de uma variável *result* que obrigatoriamente tem de estar presente na lista de *inputs*. Caso ela seja alterada durante o programa *PicoC*, então o seu valor irá ser atualizado e no final da execução do programa irá retornar o devido valor. Caso não haja referência no programa *PicoC* à variável *result*, a função retornará sempre o valor inicial associado ao *result* (que normalmente é 0).

Antes de avançar mais fundo na forma como implementamos esta solução é importante compreender as funções de avaliação *eval* e *eval2*. Estas funções avaliam expressões aritméticas e booleanas, respectivamente, utilizando um contexto (*Inputs*) que mapeia variáveis aos seus valores:

```
eval :: Exp -> Inputs -> Int
eval2 :: Exp -> Inputs -> Bool
```

Em primeiro lugar, criamos a função *evaluate*, que recursivamente vai chamar uma função auxiliar para analisar a primeira instrução que tem no programa *PicoC* que recebeu como argumento.

```
evaluate :: PicoC -> Inputs -> Int
evaluate (PicoC []) i = getResult i
evaluate (PicoC (h:t)) i = evaluate (PicoC t)
(evaluateInst h i)
```

Esta função auxiliar, *evaluateInst* é responsável por coordenar o que é suposto acontecer dependendo do tipo de instrução que recebe, e devolve a lista de *Inputs* atualizada para a função *evaluate*.

Um problema que tivemos foi ao usar a função *trace* da biblioteca *Debug.Trace*. Utilizamos esta função para dar print e devolver a lista de *Inputs* ao mesmo tempo. No entanto quando fomos testar esta implementação, os *prints* estavam a sair na ordem contrária ao suposto. Assim, o grupo viu-se forçado a encontrar uma implementação diferente da função *trace*, para resolver o problema.

VII. Programas

Para testar o código que criamos, tivemos de escrever programas *PicoC* e testes para os mesmos.

Inicialmente criamos os seguintes 3 programas:

- **Programa1:** Calcula 2^r, recebe r,g,result como *Inputs*

```
result = 1;
while (g < r){
    result = result * 2;
    g = g + 1;
}
```

- **Programa2:** *If then Else* simples. Recebe *r*, *g*, e *result* como *Inputs*.

```
if(g + 5 > n) {
  result = g;
} else {
  result = n;
}
```

- **Programa3:** Soma 1 ao *result* caso *g>5*, e soma *r* ao *result* caso *g<5*. Recebe *n*, *g*, *result* como *Inputs*

```
while(g < 10) {
  print(g);
  if(g > 5) {
    print("Entrei na condicao if(g>5)");
    result = result + 1;
  } else {
    print("Entrei na condicao else(g<=5)");
    result = result + r;
  }
  g = g + 1;
}
```

- **Programa Extra:** Por fim, criamos um programa mais complexo que usa todas as instruções que criamos, para observar o seu funcionamento em diversos casos de teste.

```
x = 0;
y = 5;
while(x < 20) {
  print(x);
  if(x >= 10) {
    print("Entrou na condicaoo if(x >= 10)");
    y = y * 2;
    if(y == 20) {
      print("Entrou na condicao if(y == 20)");
      result = result + 1;
    } else {
      print("Entrou na condicao else(y != 20)");
      result = result + z;
    }
  } else {
    print("Entrou na condicao else(x < 10)");
    y = y - 1;
    while(y < 0) {
      print("Entrou no loop interno");
```

```
      y = y + 2;
      print(y);
    }
  }
  x = x + 1;
  // Comentario de teste
}
print("Final do programa");
print(result);
```

Para criar os testes seguimos uma estrutura similar para todos os programas. Abaixo, encontram-se representados os diferentes *Inputs* e Testes que usamos para o programa1.

```
teste11 :: Inputs
teste11 = [("g",0),("result",0),("r",3)]
teste12 :: Inputs
teste12 = [("g",0),("result",0),("r",4)]
teste13 :: Inputs
teste13 = [("g",0),("result",0),("r",5)]
teste14 :: Inputs
teste14 = [("g",0),("result",0),("r",0)]

testSuitePrograma1 :: [(Inputs,Int)]
testSuitePrograma1 = [(teste11,8),(teste12,16),
(teste13,32),(teste14,1)]

runTestSuitePrograma1 = runTestSuite programa1
testSuitePrograma1
```

VIII. Mutações

Agora vamos abordar o desenvolvimento do tópico das mutações. Uma mutação é uma alteração que faz com que o programa e os resultados que o mesmo retorno fiquem errados.

A forma como fazemos a alteração no programa é bastante parecida à forma como aplicamos as otimizações. No entanto, em vez de percorrermos a árvore das otimizações toda à procura de possíveis alterações, simplesmente alteramos uma só parte da função à sorte, através do uso do *once_RandomTP*.

A lista de mutações está representada na função *mut3* que, dado uma expressão, devolve o que lhe deve acontecer.

```
mut3 :: Exp -> Maybe Exp
mut3 (Add e i) = Just (Sub e i)
mut3 (Mul e i) = Just (Div e i)
mut3 (Sub e i) = Just (Add e i)
mut3 (Const e) = Just (Add (Const e) (Const 10))
mut3 (Great e i) = Just (MinorEqual e i)
mut3 (GreatEqual e i) = Just (Minor e i)
mut3 (Minor e i) = Just (GreatEqual e i)
mut3 (Equals e i) = Just (Dif e i)
mut3 (Dif e i) = Just (Equals e i)
mut3 (Not e) = Just e
mut3 (And e i) = Just (Or e i)
mut3 (Or e i) = Just (And e i)
mut3 Verdadeiro = Just Falso
mut3 Falso = Just Verdadeiro
mut3 _ = Nothing
```

IX. Instrução PRINT

Para a implementação do *print* primeiro começamos por extender o nosso datatype *Inst* e *Exp*. Esta extensão permite, primeiramente, desenvolvermos programas PicoC com instruções de *print*, e de seguida, aliada à utilização da biblioteca *Debug.trace* facilita o debug quando estamos a correr a função *evaluate*. Por último, mas não menos importante, durante a instrumentação, mesmo que o programa PicoC não tenha qualquer tipo de instrução *PRINT*, a nossa função consegue colocar automaticamente prints da expressão que acabou de passar durante a execução do programa.

X. Fault Localization

Após termos 3 programas com os seus respetivos testes e a sua respetiva mutação, avançamos para a aplicação do algoritmo *Spectrum-Based Localization*

- **Programa 1**

	result = 1	while (g>=r)	result= result^ 2	g=g+1	Final
t1	1	1	0	0	1
t2	1	1	0	0	1
t3	1	1	0	0	1
t4	1	1	1	1	1
%	1	1	1/4	1/4	*N/A*

- **Programa 2**

	if(g+5+10)	result=g	else	result=n	Final
t1	1	1	*N/A*	0	1
t2	1	1	*N/A*	0	0
t3	1	1	*N/A*	0	0
t4	1	0	*N/A*	1	0
t5	1	1	*N/A*	0	1
%	2/5	2/4	*N/A*	0	*N/A*

- **Programa 3**

	while (g< 10)	if (g <=5)	result = result +1	else	result = result +r	g= g+1	Final
t1	1	1	1	*N/A*	1	1	0
t2	1	1	1	*N/A*	1	1	1
t3	1	1	1	*N/A*	1	1	1
t4	1	1	0	*N/A*	1	1	1
%	3/4	3/4	1/2	*N/A*	3/4	3/4	*N/A*

XI. Conclusão

Em suma, este trabalho proporcionou uma experiência abrangente em diversos aspectos do desenvolvimento de linguagens de programação, desde a concepção da gramática até à implementação prática de um interpretador eficiente. Através deste projeto, adquirimos também um conhecimento mais profundo sobre *haskell* e as suas bibliotecas, algo com que antigamente podiamos não estar tanto à vontade.

O facto de ter sido um trabalho desenvolvido ao longo de um semestre inteiro, e não tudo feito 2 semanas antes, fez com que o trabalho em geral se tornasse menos cansativo e maçudo.

Além disso, por ser um trabalho no qual se pode investir tempo infinito, dado que há sempre a possibilidade de adicionar novas funcionalidades, o futuro do nosso projeto é promissor. Entre as possibilidades de melhorias, incluem-se a adição de novas instruções, o aprimoramento dos geradores, e a implementação de declarações obrigatórias de variáveis antes do uso.